

AD-A117 049

MARYLAND UNIV COLLEGE PARK COMPUTER VISION LAB  
CELLULAR ARCHITECTURES FOR PATTERN RECOGNITION.(U)  
APR 82 A ROSENFELD

F/6 9/2

UNCLASSIFIED

TR-1151

AFOSR-77-3271  
AFOSR-TR-82-0559

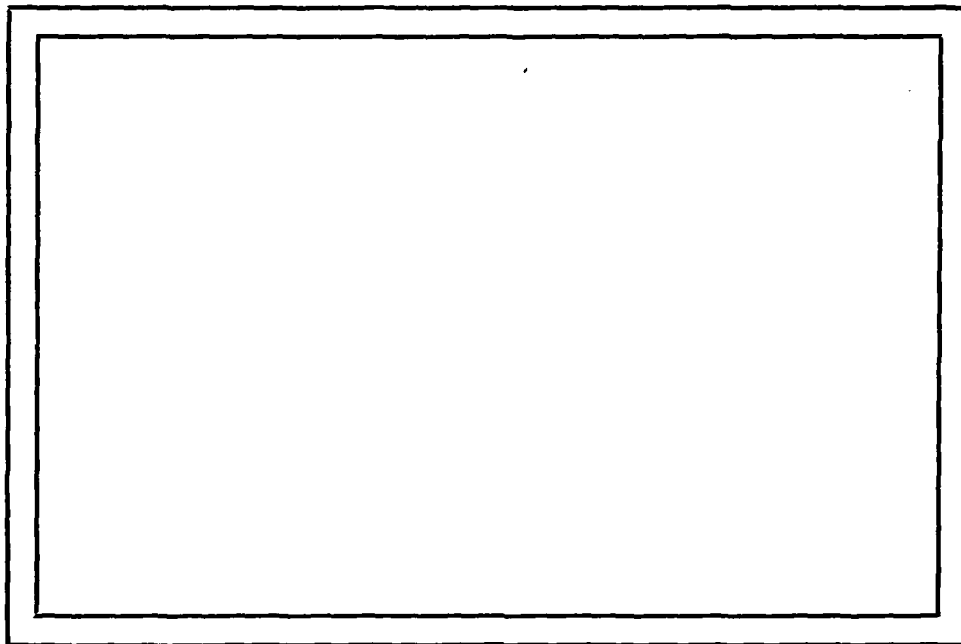
NL

1 09 1  
40 A  
1 70 4 3

END  
DATE  
FILMED  
08-82  
DTIC

10

AFOSR-TR- 82 - 0559



AD A117049

DTIC FILE COPY



DTIC  
SELECTED  
JUL 20 1982  
F

UNIVERSITY OF MARYLAND  
COMPUTER SCIENCE CENTER

COLLEGE PARK, MARYLAND  
20742

Approved for public release  
distribution unlimited.

82 07 19 161

TR-1151  
AFOSR-77-3271

April 1982

Accession For		
NTIS GRA&I	<input checked="" type="checkbox"/>	
NO TAB	<input type="checkbox"/>	
Unannounced	<input type="checkbox"/>	
Justification		
Distribution/		
Availability Codes		
Avail and/or		
Special		
A		

CELLULAR ARCHITECTURES  
FOR PATTERN RECOGNITION

Azriel Rosenfeld

Computer Vision Laboratory  
Computer Science Center  
University of Maryland  
College Park, MD 20742



ABSTRACT

Cellular computers - arrays of processors each of which can communicate with their neighbors - were proposed nearly 25 years ago as natural computational structures for image processing and recognition. Several such processor arrays, containing up to 128 by 128 processors, have actually been constructed. This article reviews basic methods of image processing using cellular arrays, and also discusses some possible extensions and generalizations.

The support of the U.S. Air Force of Scientific Research under Grant AFOSR-77-3271 is gratefully acknowledged, as is the help of Janet Salzman in preparing this paper.

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)  
NOTICE OF TRANSMITTAL TO DTIC  
This technical report has been reviewed and is  
approved for public release IAW AFR 130-12.  
Distribution is unlimited.  
MATTHEW J. KEMPER  
Chief, Technical Information Division

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM	
1. REPORT NUMBER <b>AFOSR-TR- 82-0559</b>	2. GOVT ACCESSION NO. <b>AD-A117049</b>	3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) CELLULAR ARCHITECTURES FOR PATTERN RECOGNITION		5. TYPE OF REPORT & PERIOD COVERED TECHNICAL	
7. AUTHOR(s) Azriel Rosenfeld		6. PERFORMING ORG. REPORT NUMBER TR-1151	
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Vision Laboratory, Computer Science Center, University of Maryland College Park MD 20742		8. CONTRACT OR GRANT NUMBER(s) AFOSR-77-3271	
11. CONTROLLING OFFICE NAME AND ADDRESS Mathematical & Information Sciences Directorate Air Force Office of Scientific Research Bolling AFB DC 20332		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS PE61102F; 2304/A2	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		12. REPORT DATE APR 82	
		13. NUMBER OF PAGES 25	
		15. SECURITY CLASS. (of this report) UNCLASSIFIED	
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.			
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)			
18. SUPPLEMENTARY NOTES			
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Image processing; pattern recognition; parallel processing; cellular arrays; cellular automata.			
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Cellular computers - arrays of processors each of which can communicate with their neighbors - were proposed nearly 25 years ago as natural computational structures for image processing and recognition. Several such processor arrays, containing up to 128 by 128 processors, have actually been constructed. This article reviews basic methods of image processing using cellular arrays, and also discusses some possible extensions and generalizations.			

## 1. Introduction

Nearly 25 years ago, Unger<sup>1,2</sup> suggested that a natural computer architecture for image processing and recognition would be a two-dimensional array of processing elements. Ideally, in this approach, each of the processors is responsible for one pixel (= one element of the image), with neighboring processors responsible for neighboring pixels. Thus, using hard-wired communication between neighboring processors, it becomes possible to perform local operations on the image, or to detect local image features, in parallel, with every processor simultaneously accessing its neighbors and computing the appropriate function on its neighborhood.

Over the past two decades, several machines embodying this concept have been constructed. The first of these was ILLIAC III<sup>3</sup>, which made use of a 36-by-36 processor array (by contrast, the later ILLIAC IV used only an 8-by-8 array); it was intended for the analysis of "events" in nuclear bubble chamber images by examining 36-by-36 "windows" of the images. Later machines, such as CLIP<sup>4</sup>, DAP<sup>5</sup>, and MPP<sup>6</sup>, use arrays of up to 128 by 128 processors, and must also be applied blockwise to larger images.

This article reviews the basic techniques of image processing using two-dimensional arrays of processors, or "cellular arrays". It also discusses various extensions and generalizations of the cellular array concept and their possible implementations and applications.

## 2. Cellular arrays

A cellular array is a two-dimensional array, usually assumed to be rectangular in shape, of processors ("cells") each of which can directly communicate with its neighbors in the array. Here "neighbors" can be defined in various ways; we shall assume, for simplicity, that each cell is connected to its four horizontal and vertical neighbors. Note that the cells on the borders of the array have only three neighbors each, and the cells in the four corners of the array have only two each. It is assumed that a cell knows which of its neighbors is which, i.e., it can send a different message to each neighbor, and when it receives messages from them, it knows which message came from each neighbor.

We will not be concerned here with hardware aspects, but will treat the cells and their intercommunication on an abstract level. It should be pointed out, however, that the two-dimensional array structure is very appropriate in terms of layout on a set of (two-dimensional) chips. The connections between cells and their neighbors do not have to cross one another, and connections between two chips are needed only for the cells along the chip borders, so that they can be handled by connectors located around the borders. Figure 1 shows, schematically, a portion of a cellular array.

To use a cellular array for image processing, we give each cell the value of an element of the image (a pixel) as input data. If the cellular array is smaller than the image, this

means that we must process the image a block at a time, and keep track of what happens where the blocks meet or overlap (Figure 2a). Alternatively, if the cells have enough storage capacity, we can give each of them a block of image pixels as input; neighboring cells must then exchange information about all the pixels located on the borders of their blocks (Figure 2b). In the cellular arrays actually built up to now, each cell has very little memory (e.g., at most 1024 bits), so this alternative would not be practical. We shall assume here, for simplicity, that each cell has only a single pixel as input, but we shall not discuss how to handle the "seams" between blocks of the image if it is necessary to process the image blockwise.

The principal advantage of using a cellular array for image processing is that the processors can all operate in parallel on the neighborhoods of their pixels, so that local operations can be performed on the entire image in an amount of time that does not grow with the image size. As a very simple example, suppose we want to average each pixel with its neighbors. We can do this by having every processor execute the following sequence of instructions, where each instruction is carried out by all processors in parallel:

1. Add own value into register (initially register is zero)
2. Read value of north neighbor and add into register
- 3-5. Analogous to step (2) for east, south, and west neighbors

6. Divide contents of register by 5\*
7. Replace own value by contents of register

Given a suitable repertoire of such instructions, a wide variety of local image operations can be carried out in parallel. If a cell has too little memory to store a program composed of such instructions, the sequence of instructions to be carried out can be broadcast to all the cells. (In some of the existing cellular array machines, in fact, the cells have very little memory, and they operate on bits rather than on integers or real numbers; such a machine would read the neighbor values in a bit at a time, and perform the addition by a series of bit-wise logical operations.)

By contrast, a conventional computer, having only a single processor, can only perform the averaging process for one pixel at a time. Thus the total time required to do local image averaging, or any other local operation, on a conventional computer is proportional to the image area. In other words, for an  $n$ -by- $n$  image, using a cellular array increases the amount of hardware needed by a factor proportional to  $n^2$ , but it decreases the time required by a similar factor.

---

\*We ignore here the special treatment required for the pixels on the borders of the image. The results obtained will be meaningful only for non-border pixels.

Unfortunately, the time required to perform local operations is not the whole story; time is also needed to input the image into a cellular array and to output the processed image. In practice, the rows of the image can be shifted into the rows of the array in parallel, so that the total time for readin or readout of an  $n$ -by- $n$  image is proportional to  $n$  rather than to  $n^2$ ; but this still grows with the image size, though not as quickly. Similar problems arise if we want to output information about the image, e.g., if we want to count the number of occurrences of a given value (as in histogramming); messages representing these occurrences must be shifted to where they can be (counted and) output, which still takes time proportional to  $n$ . Thus the speedup resulting from the use of a cellular array ( $n^2$  processors) is not as great as it might seem at first glance (order( $n$ ), not order( $n^2$ ), faster than a single processor).

### 3. Cellular array algorithms

A wide variety of image processing algorithms appropriate for cellular arrays have been developed; some of them are straightforward, but others are very unobvious. For example, algorithms exist<sup>7,8</sup> that will count, in time proportional to  $n$ , the number of connected components of 1's in an  $n$ -by- $n$  array of 1's and 0's (a two-valued image), or that will assign a unique label to the pixels belonging to each such component; these algorithms are quite nontrivial (see below).

In much of the theoretical work on the computational power of cellular arrays, it has been assumed that the amount of memory in a cell remains bounded no matter how many cells there are. With this assumption, the cells can be regarded as finite-state machines, and the cellular array is thus a two-dimensional "bounded cellular automaton" (BCA). Efficient algorithms for BCA's have been extensively studied; in fact, three books on BCA's appeared at the end of the 1970's<sup>9,10,11</sup>.

From a practical standpoint, the bounded-memory assumption is unnecessarily restrictive; it implies, in particular, that a cell does not have enough memory to store the coordinates of its position in the array! Realistically, when we are able to build very large cellular arrays, we will certainly be able to give each cell a modest amount of memory, say growing logarithmically with the array size. (Note that this is now sufficient for a cell to store its coordinates, which are  $k$ -bit

numbers for an array of size  $2^k$  by  $2^k$ .) When we do this, it simplifies the design of cellular array algorithms for many basic tasks<sup>12</sup>. A variety of such algorithms are sketched in the following paragraphs.

a. Local operations (local property computation)

A local image property is one whose (output) value at a given pixel is a function of the (input) values of only a (small) set of the pixel's neighbors (possibly including the pixel itself). A cellular array can compute a local property in parallel at each pixel by shifting the values of the needed neighbors until they reach the processor corresponding to the pixel; once it has received them, it computes the desired function. As already pointed out, the amount of time required to do this in parallel is independent of the image size. Local properties are very widely used in image processing for such purposes as smoothing, deblurring, edge detection, texture analysis, etc.

b. Value counting (histogramming)

To count the number of occurrences of a given value in an image, we must send messages (e.g., 1's) representing the occurrences to a central counter where they can be summed. For example, we can shift the 1's leftward in each row of the image and sum them in the leftmost cell. We can then shift the sums in these leftmost cells upward and sum them in the

upper left cell. If we assume that addition of two numbers takes unit time, then the total time required for the shifting and summing is proportional to the width + height of the image, i.e., of order  $n$  [ $O(n)$ ] for an  $n$ -by- $n$  image. This method can be used to construct the gray level histogram of an image in  $O(n)$  time. Other types of image statistics, such as gray level cooccurrence matrices, can also be computed in  $O(n)$  time.

#### c. Moments and transforms

To compute the value of a given moment of the image, or of a given coefficient in a transform (Fourier, Hadamard, etc.) of the image, we must multiply the image pixelwise by the appropriate basis matrix and sum the results. The basis matrix values can be computed by or "broadcast" to the pixels; e.g., in the Fourier case, we start with the appropriate root of unity, and raise it to a higher power each time we shift it. Once they have been computed or received, the multiplication is done in a single parallel step; and the results can then be shifted and summed. Evidently, the broadcasting and summing steps require  $O(n)$  time.

#### d. Connected components

Given an array of 0's and 1's, a local "shrinking" process can be defined<sup>7</sup> that collapses each component of 1's into a single 1 (which then disappears) in time proportional to the diameter of the component's circumscribing rectangle. To count the

components of 1's, we change the singleton 1's, as they disappear, into special marks, which we then shift to the upper left corner of the image and count. The entire process takes  $O(n)$  time. Labelling the components is more complicated. We first identify a distinguished pixel in each component, which can be done in  $O(n)$  time, and we assign a unique label to each distinguished pixel (e.g., its coordinates). Finally, we construct a minimal spanning tree of each component, rooted at the distinguished pixel<sup>13</sup>, and broadcast each label to every node of its tree; this takes time proportional to the tree height.

#### e. Region representations

An array of 0's and 1's can be specified in several different ways. Each row can be represented by "run length code" which gives the successive lengths of the runs of 0's and 1's (or vice versa) that comprise the row. Each connected component can be represented by the "chain codes" of its borders (defining the sequence of moves required to travel around each border), together with the coordinates of a starting point on each border. The set of 1's can be represented as a union of maximal blocks (e.g., upright squares); the centers of these blocks turn out to be the pixels where "chessboard distances" from the set of 0's are local maxima. It is not difficult to define cellular array algorithms<sup>12</sup> that construct each of these

representations from a given array of 0's and 1's, or that reconstruct the array from the representation. The time required depends on the image diameter (i.e.,  $O(n)$ ) and on the compactness of the representation; but one should not use such a representation unless it is in fact compact.

f. Region property computation

It is straightforward to compute properties of a region such as its area and perimeter by using a minimal spanning tree to sum the number of pixels (or border pixels) in the region. Similarly, the height and width of a region can be computed by using the tree to determine the highest and lowest x and y coordinates of the pixels in the region. The "thickness" of a region is twice the greatest distance from any region pixel to the border of the region; the distances can be computed by propagating a signal from every border pixel, and incrementing a counter at each pixel until the signal reaches it. Region shape properties such as compactness and elongatedness can be determined in terms of area, perimeter, and thickness; and shape complexity can be measured as the sum of the (absolute) angles defined by successive triples of border pixels. The convexity of a region is more difficult to determine using a cellular array<sup>14</sup>.

#### 4. One-dimensional cellular arrays ("cellular strings")

Two-dimensional cellular arrays are still quite expensive to build; the largest ones now in existence are only 128 by 128, and do not have very much memory per processor. One-dimensional cellular arrays are much more economical, and could be used for parallel processing of various types of waveforms. In the following paragraphs we consider two ways of using one-dimensional cellular arrays ("cellular strings") for image-related tasks.

As mentioned earlier, a region border (or a curve) can be represented by specifying the sequence of moves (from neighbor to neighbor) required to traverse it; this sequence is called a chain code. Cellular strings can be used to efficiently derive information about curves or regions, given the chain codes of the curves or of the regions' borders<sup>15</sup>. [Similar remarks apply if the curves or borders are specified as polygons having sides of arbitrary length, rather than as sequences of unit moves. Such representations are extensively used in digital cartography.] For example, the following can be determined in time proportional to the number of links in the chain (or chains, if two curves are involved): Whether the curve is a digitized straight line; the points at which it touches or intersects itself; and whether one (non-selfintersecting) curve surrounds another. Similarly, the chain code(s) of the border(s) of the union or intersection of two regions can be constructed; and various types of polygonal approximations to a curve can be constructed.

A cellular string can also be used to process a two-dimensional image by scanning it row by row and operating in parallel on each row<sup>16</sup>. It can perform local operations by storing several rows in order to obtain the needed neighbors' values; note that this now takes  $O(n)$  time for an  $n$ -by- $n$  image, because of the need for a row-by-row scan. On the other hand, it can still perform counting operations in only  $O(n)$  time, e.g., summing the 1's in each column as it scans, and then shifting and summing the column sums when it reaches the bottom row. Some tasks probably require more than  $O(n)$  time, but many basic operations require only  $O(n)$  time at the cost of only  $O(n)$  hardware, thus providing an attractive alternative to the  $O(n^2)$  hardware needed in a two-dimensional cellular array.

## 5. "Cellular hypercubes" and "cellular pyramids"

In this section we suggest some extensions of the cellular array concept that yield a substantial increase in the speed at which counting tasks can be performed.

One way of achieving this speed increase is to allow each cell to communicate not just with its immediate neighbors, but also with cells at distances 2,4,8,... from it (Figure 3). For an  $n$ -by- $n$  array, this requires  $O(\log n)$  connections for each cell, which is not an unreasonable number in view of the fact that we have already allowed the amount of memory in a cell to be  $O(\log n)$ . (It is not obvious, though, how these connections might be physically realized on a two-dimensional chip; we ignore this implementation issue here.) The resulting network of cells is similar to a  $(\log n)$ -dimensional hypercube, in which any cell can be reached from any other cell in  $O(\log n)$  moves from neighbor to neighbor. It is clear how this allows counting tasks to be carried out in  $O(\log n)$  time<sup>17,18</sup>.

An alternative approach<sup>19</sup> allows the number of connections to a cell to remain bounded, but at the cost of increasing the number of cells by a factor of 2 in the one-dimensional case or of  $1\frac{1}{3}$  in the two-dimensional case. In one dimension, we use a "stack" of cellular strings, each half the length of the preceding one. This yields an exponentially tapering "triangle" in which the total number of cells is  $n + \frac{n}{2} + \frac{n}{4} + \dots < 2n$ . Here each cell is connected to its two "brothers" in its own string,

and also to two "sons" in the next larger string (if any) and to a "father" in the next smaller one (if any), as shown in Figure 4; thus a general cell has five neighbors. Note that this connection structure can be easily laid out on a two-dimensional chip. [In two dimensions, analogously, we would use a stack of cellular arrays, each one-quarter the area of the preceding one, yield an exponentially tapering "pyramid" in which the total number of cells is  $n + \frac{n}{4} + \frac{n}{16} + \dots < \frac{4n}{3}$ . Here, we would connect each cell to its four "brothers" in its own array, to four "sons" in the next larger array (if any), and to a "father" in the next smaller array (if any) - a total of nine neighbors. Unfortunately, it is not clear how to lay out these connections on a chip.] This scheme too allows counting in  $O(\log n)$  steps, since cells can pass their counts to their fathers, so that the total count is obtained at the apex of the triangle (or pyramid), which has height  $\log n$ . Note that counting requires only the son/father connections, not the brother/brother connections, and the resulting complete binary tree (or quadtree, in the two-dimensional case) can in fact be laid out on a chip. However, we need the brother/brother connections in order to perform local operations, so that layout still poses a problem.

## 6. "Cellular graphs"

Cellular strings, arrays, triangles, pyramids, etc. are all composed of cells, each of which is connected to a fixed set of neighbors. More generally, one can consider "cellular graphs" in which the neighbor relationship is arbitrary, subject to the restriction that each cell has bounded degree, i.e., a bounded number of neighbors (or perhaps degree that grows logarithmically with the number of cells, as in the case of a hypercube). Generalizations of basic image processing algorithms, including local property computation, counting, connected component analysis, etc. can be defined for such cellular graphs operating on graph-structured input<sup>20,21,22</sup>.

Cellular graphs that have fixed graph structures are of limited interest, unless we have many sets of input data to be processed that all have the same structure. In image processing, various types of graph structures do arise - e.g., when we segment an image into regions, we can regard these regions as the nodes of a graph, with adjacency between regions defining the neighbor relationship on the nodes. However, these graphs differ from image to image, and they even vary in the course of processing a single image as we modify the segmentation by merging or splitting regions. Thus it would be of greater interest to consider cellular graphs in which the initial graph can be defined arbitrarily and can then modify itself in the course of a computation.

A class of such "reconfigurable" cellular graphs has in fact been studied<sup>23,24</sup>. It has been shown how such a cellular graph can be initially configured to represent a given segmentation of a given image, e.g., in terms of its region adjacency graph, and this graph can modify itself (and recompute the properties of the regions) as regions merge and split. Note that the region adjacency graph can have very high degree, since many regions can be adjacent to a given region. We can obtain a graph of bounded degree by letting the nodes represent boundary segments where pairs of regions meet, and letting adjacencies of these segments define the neighbor relationship. An alternative is to represent the segmented image by a "quadtree", defined by recursive subdivision into quadrants, sub-quadrants, ..., until blocks of constant value are obtained. In this approach too, we can construct a "cellular quadtree" corresponding to the quadtree of the image, and use it to compute the region properties<sup>25</sup>.

Reconfigurable cellular graphs allow region-level image processing operations to be carried out in parallel, by assigning cells to the regions (or other pieces of the segmented image) and modifying the neighbor relationship on the cells, in parallel, as the segmentation is modified. The parallelism involved here is not as great as in pixel-level image processing, since there are relatively few regions (typically several

hundred in a conservatively segmented image) compared to the original number of pixels. However, even this degree of parallelism may be very useful when it comes to tasks involving combinatorial search, e.g., looking for subgraphs of the given graph that are isomorphic to a given "model" graph.

Some types of (fixed-structure) cellular graphs can be implemented in hard-wired form (arrays and trees, for example), but when it comes to reconfigurable cellular graphs, a different approach is evidently needed. Let us think of the cells as communicating via some standard type of interprocessor communication network. The pairs of cells that can communicate directly with one another at a given moment define the neighbor relationship of the cellular graph at that moment. As an example, consider the ZMOB multiprocessor;<sup>26</sup> it consists of 256 Z80A microprocessors that communicate with each other via a fast bus. Evidently, ZMOB can simulate a reconfigurable cellular graph having up to 256 cells. It can also be used for image processing at the pixel level, by assigning blocks of pixels to the processors (which have substantial amounts of memory - up to 64k bytes each) and allowing them to communicate as necessary<sup>27</sup>. Thus ZMOB should be useful for both pixel-level and region-level image processing and recognition.

## 7. Concluding remarks

Cellular arrays are a classical model for image processing and recognition at the pixel level. Such arrays are now being built in reasonable sizes, but they are still quite costly, and are limited in speed for many tasks due to communication delays. Simpler architectures, such as cellular strings, can be built at low cost today, and can be used for a variety of practical tasks, e.g., chain code or waveform processing, or "serial/parallel" row-by-row processing of two-dimensional images. (Modifications to the basic cellular array, such as the cellular hypercube or pyramid, could overcome some of the communication bottlenecks, but would be harder to implement.) Reconfigurable "cellular graphs" can be used for image processing at the region level; they require relatively small numbers of cells, and can be simulated by multi-microprocessor systems such as ZMOB that permit flexible interprocessor communication.

In the 25 years since cellular arrays were first proposed for parallel image processing, hardware technology has evolved to the point that such special-purpose architectures can finally be built at reasonable cost. This will lead to major increases in the computer power available for image processing, which should in turn lead to the development of algorithms that are much more powerful than those currently available.

## References

1. S. H. Unger, "A computer oriented toward spatial problems", Proc. IRE, vol. 46, pp. 1744-1750, 1958.
2. S. H. Unger, "Pattern detection and recognition," Proc. IRE, vol. 47, pp. 1737-1752, 1959.
3. B. H. McCormick, "The Illinois pattern recognition computer - ILLIAC III", IEEE Trans. Electronic Computers, Vol. 12, pp. 791-813, 1963.
4. M. J. B. Duff, D. M. Watson, T. J. Fountain, and G. K. Shaw, "A cellular logic array for image processing", Pattern Recognition, vol. 5, pp. 229-247, 1973.
5. P. Marks, "Low level vision using an array processor," Computer Graphics Image Processing, vol. 14, pp. 281-292, 1980.
6. K. E. Batchner, "Design of a massively parallel processor," IEEE Trans. Computers, vol. 28, pp. 836-840, 1980.
7. S. Levialdi, "On shrinking binary picture patterns," Comm. ACM, vol. 15, pp. 7-10, 1972.
8. S. R. Kosaraju, "Fast parallel processing array algorithms for some graph problems," Proc. 11th ACM Symp. on Theory of Computing, pp. 231-236, 1979.
9. V. Aladyev, Mathematical Theory of Homogeneous Structures and their Applications, Valgus (Tallinn, USSR), 1980.
10. A. Rosenfeld, Picture Languages: Formal Models for Picture Recognition, Academic Press (New York), 1979.
11. R. Vollmer, Algorithmen in Zellularautomaten, Teubner (Stuttgart, FRG), 1979.
12. C. R. Dyer and A. Rosenfeld, "Parallel image processing by memory-augmented cellular automata," IEEE Trans. Pattern Analysis Machine Intelligence, vol. 3, pp. 29-41, 1981.
13. W. T. Beyer, "Recognition of topological invariants by iterative arrays," MAC TR-66, MIT (Cambridge, MA), Oct. 1969.
14. J. Sklansky, L. P. Cordella, and S. Levialdi, "Parallel detection of concavities in cellular blobs," IEEE Trans. Computers, vol. 25, pp. 187-196, 1976.

15. A. Y. Wu, T. Dubitzki, and A. Rosenfeld, "Parallel computation of contour properties," IEEE Trans. Pattern Analysis Machine Intelligence, vol. 3, pp. 331-337, 1981.
16. A. Rosenfeld and D. L. Milgram, "Parallel/sequential array automata," Information Processing Letters, vol. 2, pp. 43-46, 1973.
17. R. Klette, "A parallel computer for digital image processing," Elektronische Informationsverarbeitung Kybernetik, vol. 15, pp. 237-263, 1979.
18. R. Klette, "Parallel operations on binary images," Computer Graphics Image Processing, vol. 14, pp. 145-158, 1980.
19. C. R. Dyer and A. Rosenfeld, "Triangle cellular automata," Information Control, vol. 48, pp. 54-69, 1981.
20. P. Rosenstiehl, J. R. Fiksel, and A. Holliger, "Intelligent graphs: networks of finite automata capable of solving graph problems." In R. C. Read, ed., Graph Theory and Computing, pp. 219-265, Academic Press (New York), 1972.
21. A. Wu and A. Rosenfeld, "Cellular graph automata (I and II)," Information Control, vol. 42, pp. 305-353, 1979.
22. A. Wu and A. Rosenfeld, "Sequential and cellular graph automata," Information Sciences, vol. 20, pp. 57-68, 1980.
23. A. Rosenfeld and A. Wu, "Parallel computers for region-level image processing," Pattern Recognition, vol. 15, pp. 41-50, 1982.
24. A. Rosenfeld and A. Wu, "Reconfigurable cellular computers," Information Control, in press.
25. T. Dubitzki, A. Wu, and A. Rosenfeld, "Parallel region property computation by active quadtree networks," IEEE Trans. Pattern Analysis Machine Intelligence, vol. 3, pp. 626-633, 1981.
26. C. Rieger, "ZMOB: Doing it in parallel!," Proc. Workshop on Computer Architectures for Pattern Analysis and Image Database Management, pp. 119-124, 1981.
27. T. Kushner, A. Y. Wu, and A. Rosenfeld, "Image processing on ZMOB," ibid., pp. 88-95.

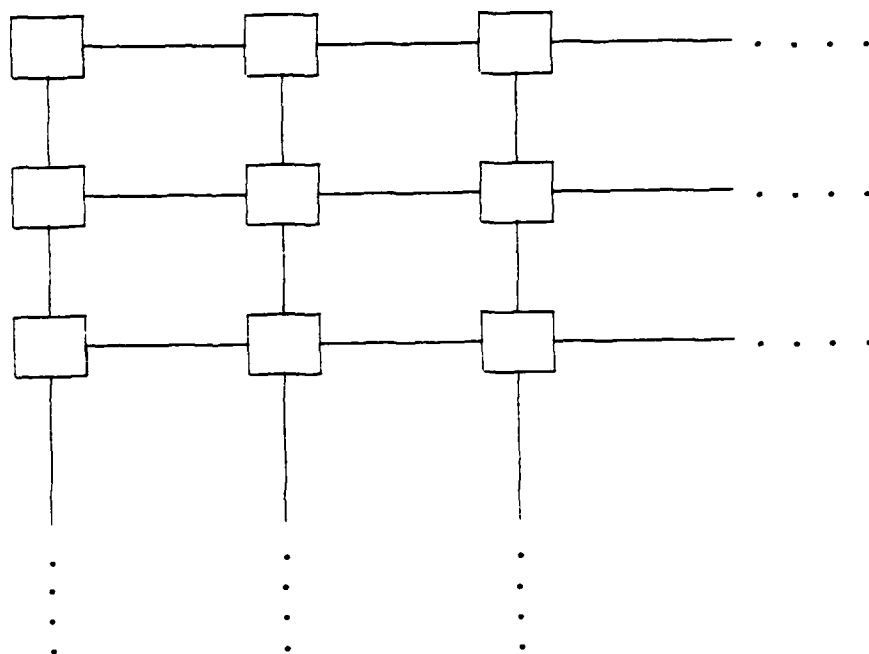
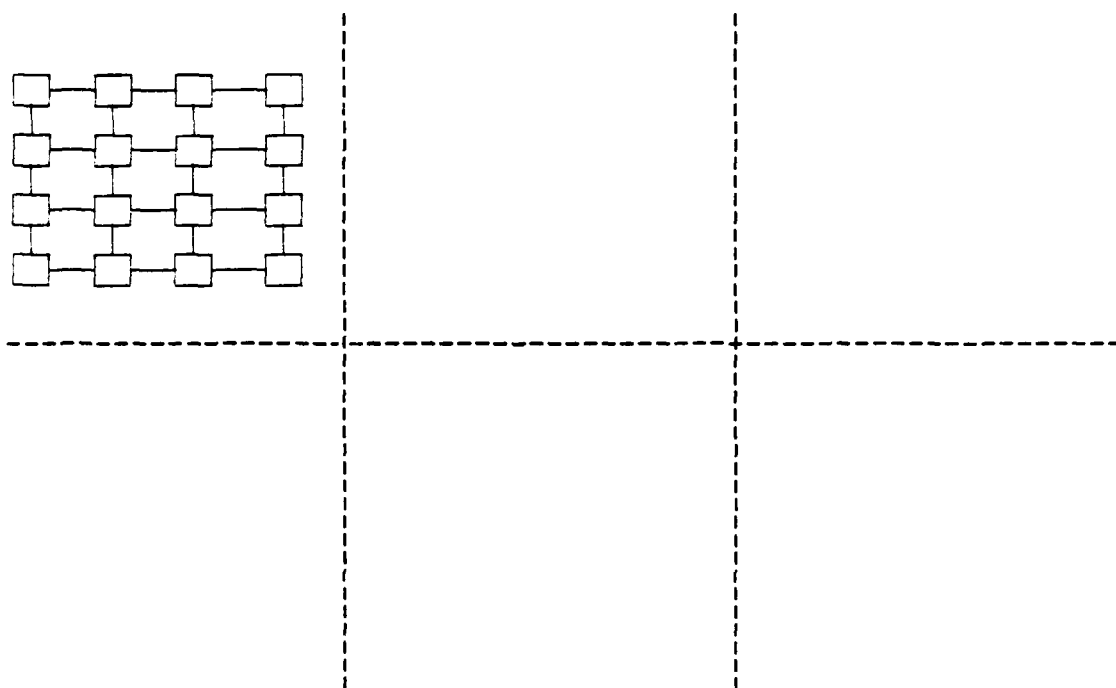
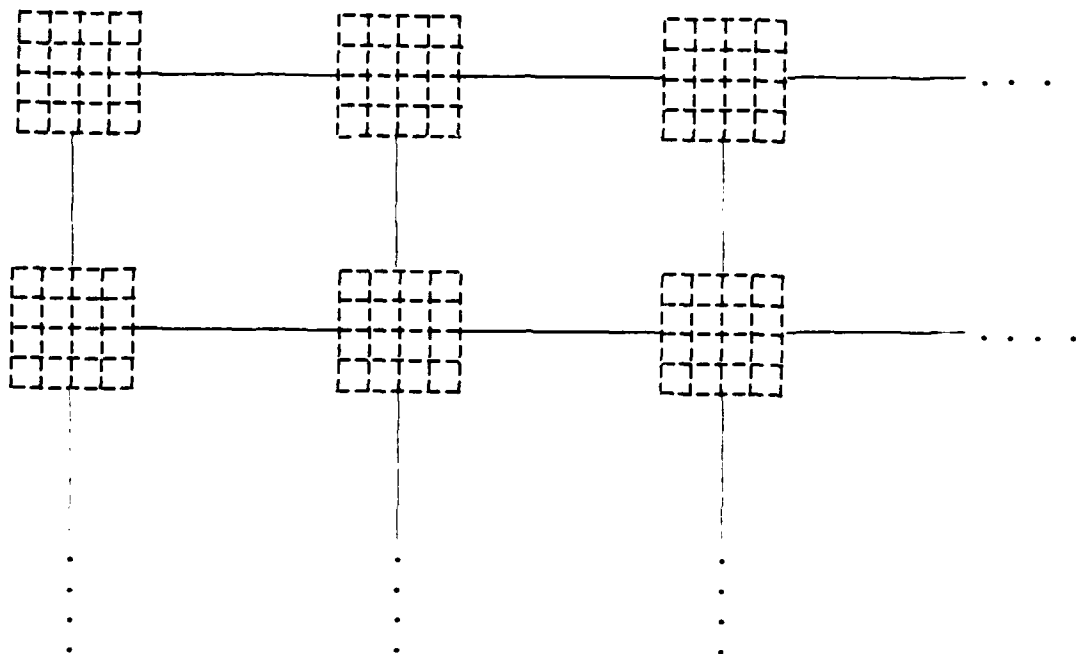


Figure 1. A two-dimensional cellular array.



(a)

Figure 2a). One way of using a small cellular array ( $m$  by  $m$ ) to process a large image ( $n$  by  $n$ ) is to process one  $m$ -by- $m$  block of the image at a time. This does not provide for communication across the "seams" where the blocks meet (dotted lines in the Figure); information about the pixels adjacent to the seams must be stored, or overlapping blocks must be used.



(b)

Figure 2b). Another way is to give each cell as input an  $(\frac{n}{m})$ -by- $(\frac{n}{m})$  block of the image; a pair of neighboring cells must now exchange information about  $\frac{n}{m}$  pairs of neighboring pixels along their common border.

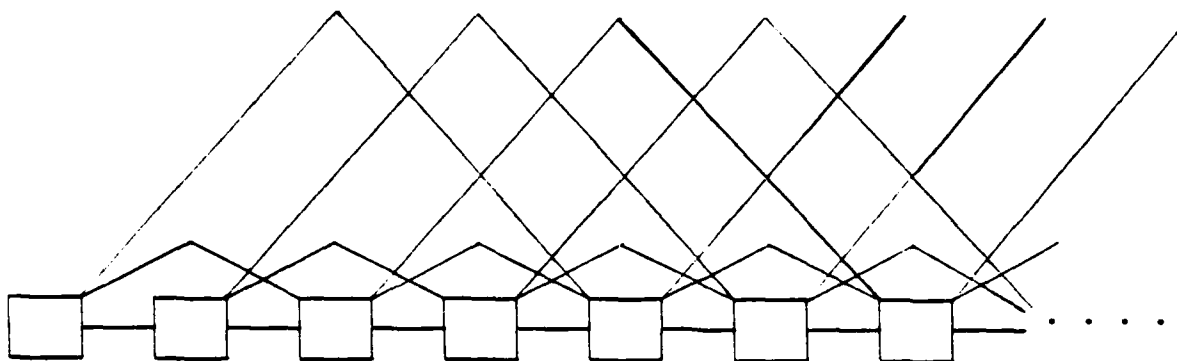


Figure 3. A one-dimensional "cellular hypercube" (only the connections to neighbors at distances 1, 2, and 4 are shown, for simplicity).

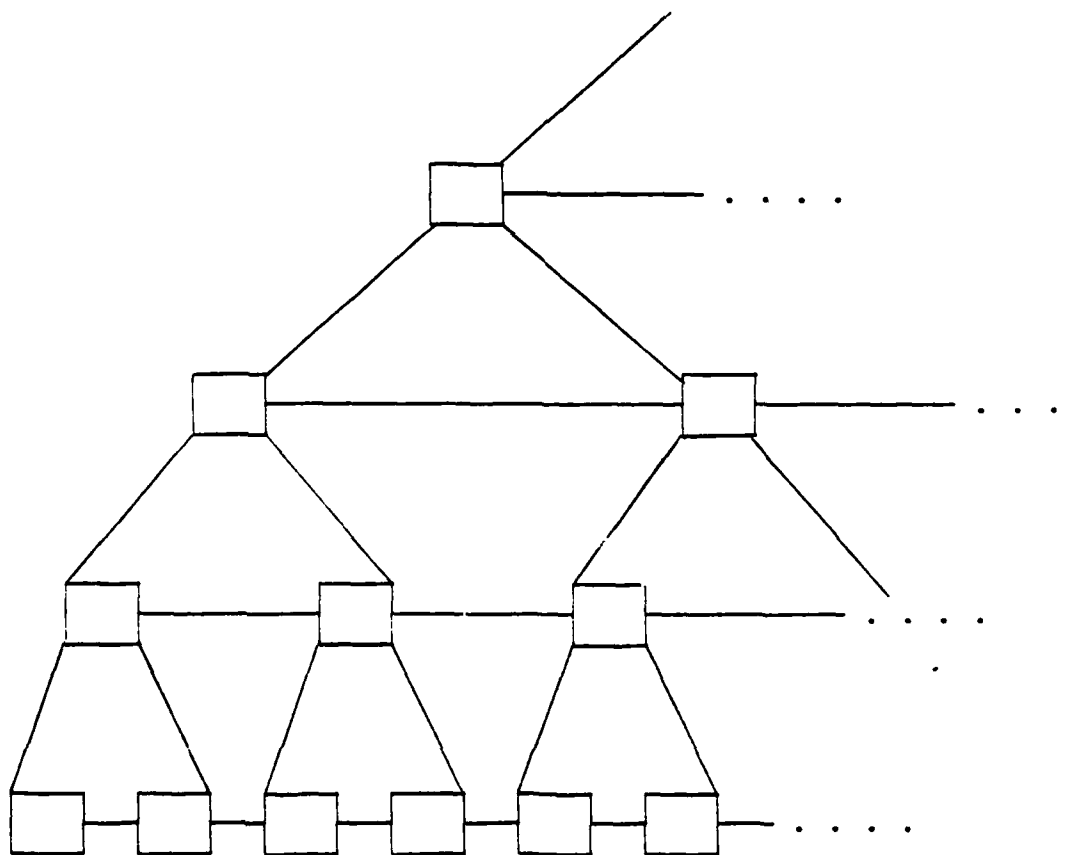


Figure 4. A "cellular triangle".

FILM  
8-